Universitat de Lleida

TREBALL FINAL DE GRAU

Estudiant: **José Manuel Broto Vispe**

Titulació: Grado en Ingeniería Informática

Títol de Treball Final de Grau: Adding new functionality to COPASI, a software for simulation of biological circuits in systems biology

Director/a: Jordi Bartolome Tomas, Rui Carlos Vaqueiro de Castro Alves

Presentació
Mes: Julio.

Any: 2019.

Adding new functionality to COPASI, a software for simulation of biological circuits in systems biology

Jose Manuel Broto[1], Jordi Bartolome[2], and Rui Alves[3]

[1] Degree in Computer Engineering student, Universitat de Lleida, Lleida, Spain
jbv10@alumnes.udl.cat
[2] Dept. of Computer Science and INSPIRES, Universitat de Lleida, Lleida, Spain
jordib@diei.udl.cat
[3] Dept. of Basic Medical Sciences and IRBLleida, Universitat de Lleida, Lleida, Spain
ralves@cmb.udl.cat

Abstract. Currently there are many software available to build and simulate biological systems, but almost none of them is capable of implementing functions like the Power Laws, the Saturating Cooperative and the Saturating. The objective of this project was to add these functions to the COPASI software, so it can be more profitable for its users. COPASI is one of the most popular systems biology standalone programs currently available. It's is programmed in C++ and Qt (GUI). To do these modifications, we had to analyze and understand the COPASI source code, and then we added the desired modifications. This was not a trivial modification, as these three functions share a common problem: they are declared differently depending on the number of substrates and modifiers of the reaction in which is selected. Usually, functions are defined as static functions, but these three were dynamic functions and that is why it was a complex problem to implement this modification. The objective has been fulfilled successfully and now it is more usable by the users. A comparison test between original COPASI and the modified COPASI was carried out by building the Overall Feedback model on both version. In this test it has been possible to appreciate the effectiveness of the implemented functions by decreasing the time by half in the complete creation of the model. The time has been highly reduced as in the modified version user must not implement manually any kind of function.

Keywords: COPASI · New functionality · Function · Rate Law · Power Laws · Saturating Cooperative · Saturating · Systems Biology · Software.

1 Introduction

Currently there are many softwares available to build and simulate biological systems [1] [7]. One of the software that stands out is COPASI, since it has many features such as deterministic simulation, stochastic simulation or SBML model

file support. COPASI is an open-source software based on the GEPASI software developed by Pedro Mendes in the 90s. This software application is created by the collaboration between the University of Manchester, the University of Heidelberg and the Virginia Bioinformatics Institute. It allows to create and solve mathematical models of biological processes. COPASI has three main parts. The first is the definition of models, which includes species and chemical reactions. The second includes the tasks that are to be applied on the models, such as deterministic and stochastic time course simulations or stoichiometric analysis among others. The third is the ability to export and import models to the SBML file format support. COPASI programming languages include C++, C and Qt for building the graphical user interface [6] [5].

Although COPASI has many features and is one of the best software applications in this field, it lacks some functionalities, such as the possibility of using complex functions such as the Power Laws, the Saturating Cooperative and the Saturating [2] [8]. These functions have in common that they are not static functions. They are suitable for all possible reactions. However they must be tailored for every reaction depending on the number of substrates and modifiers (functions are named Rate Laws in COPASI).

Each of these mentioned complex functions has a functional form we used to define our modifications.

The first functional form is the Power Laws function:

$$\alpha \prod_{j=1}^{m} \mathcal{X}_j^{g_j} \prod_{k=m+1}^{n} \mathcal{M}_k^{g_k} \tag{1}$$

α is a number parameter. m and n are the number of substrates and modifiers of the selected reaction. X represents all the substrates in the selected reaction. M represents all the modifiers in the selected reaction. g is an array of numeric parameters that are connected with every substrate and modifier of the selected reaction.

The second functional form is the Saturating Cooperative function:

$$\frac{V \prod_{j=1}^{m} \mathcal{X}_j^{g_j} \prod_{k=m+1}^{n} \mathcal{M}_k^{g_k}}{\prod_{j=1}^{m} (\mathcal{K}_j + \mathcal{X}_j^{g_j}) \prod_{k=m+1}^{n} (\mathcal{K}_k + \mathcal{M}_k^{g_k})} \tag{2}$$

V is a number parameter. m and n are the number of substrates and modifiers of the selected reaction. X represents all the substrates in the selected reaction. M represents all the modifiers in the selected reaction. g is an array of numeric parameters that are connected with every substrate and modifier of the selected reaction. K is also an array of numeric parameters that are connected with every substrate and modifier of the selected reaction.

The third functional form is that of the Saturating function:

$$\frac{V \prod_{j=1}^{m} \mathcal{X}_j \prod_{k=m+1}^{n} \mathcal{M}_k}{\prod_{j=1}^{m} (\mathcal{K}_j + \mathcal{X}_j) \prod_{k=1}^{n} (\mathcal{K}_k + \mathcal{M}_k)} \tag{3}$$

The Saturating function is similar to the Saturating Cooperative function. It only differs in that it doesn't use the g array of parameters.

The Copasi application allows you to create systems biology models. A model is a set of species, which are used to form reactions. Each reaction is assigned a velocity function. A reaction is formed by substrates, products and modifiers. Substrates are the input of the reaction, products are the output of the reaction and modifiers intervenes in the function velocity.

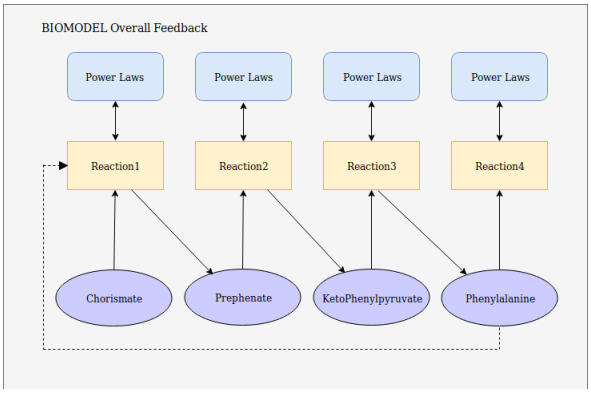

Fig. 1: Systems biology model diagram

The arrows that enter the reactions are substrates, the arrows that come out of the reactions are products and the broken arrows are modifiers.

	Reaction definition	**Kinetic function**
R1	Chorismate -> Prephenate; Phenylalanine	$\alpha * (Chorismate^{g1}) * (Phenylalanine^{g2})$
R2	Prephenate -> KetoPhenylpyruvate	$\alpha * (Prephenate^{g1})$
R3	KetoPhenylpyruvate -> Phenylalanine	$\alpha * (KetoPhenylpyruvate^{g1})$
R4	Phenylalanine ->	$\alpha * (Phenylalanine^{g1})$

Table 1: Reactions and functions definition

2 Related work

2.1 Other related software

Currently there are few biological software systems that support the native use of the Power Laws function, such as EasyModel [3], PLAS [4] or ESSYNS [9]. All software that allow user-defined kinetic functions, could use this function if it was entered manually. The point is that this process is very time consuming and error-prone. Thus we automated this process with our implementation. On the other side, there are the Saturating Cooperative and Saturating functions. We could not find any software aside from EasyModel that implements them.

To summarize, as of today, there are not many software that support this type of functions due to their implementation complexity. This is because these new functions are not static and they need to be generated according to the number of substrates and modifiers of each reaction.

2.2 Limitations and proposed contribution

The COPASI software was initially designed to use static functions in the model reactions. That means functions are not altered in any point since their creation. This is the biggest limitation on the COPASI's part that has been found at the time of carrying out the work since all the new functions that had to be incorporated in the COPASI are dynamic and vary for each reaction. Due to this fact and its complexity, few software programs have implemented these functions as described above.

At this point there were several options by which we could implement the dynamic functions. The first of which was to focus on an special function included in COPASI. This function is the Mass Action function, which implements a product of species. However, it lacked the possibility of using array parameters for each substrate and modifier. COPASI does not have implemented those arrays, so it was a major drawback. The second option and the chosen one, consisted in generating the functions dynamically at the time of selecting them in the reaction. Each time a dynamic function is generated, it is added to the general function list so COPASI can use it properly. If the required function has been already generated, user will use that one a new function will not be generated.

This way you can have the functionality of the dynamic functions (Power Laws, Saturating Cooperative and Saturating) using the standard static functions point of view.

3 Methodology

3.1 Programming language, technology, implementation and algorithm

A large part of COPASI is programmed in C++. In the C++, header files define the global variables, public and private attributes and the methods. In the implementation files is where the methods are implemented use and where the functionality of the application is. The graphic user interface has " .ui " files where all the selectors, combobox, buttons... of each window are saved. Each file of this type has an associated C++ file where in the case of a combobox it initializes or changes the data as needed. The implementation of the modifications has been done in a Linux operating system using the Sublime text editor to modify the text and to edit the graphical part of the Qt5. For the compilation of the code, the source code of the COPASI and the copasi-dependencies GitHub repository have been used, together with the ninja C++ compiler.

With regard to the implementation, we wanted to create functions dynamically and for this the CFunction class was used, which has the necessary attributes and methods to define and modify a new function. The first problem we had was to obtain the exact number of substrates and modifiers that the reaction has on which the dynamic function is going to be created. There is a method in the CChemEqInterface.cpp file that passes it through a role (Substrate, Modifier or Product) and returns the total number of substrates or modifiers (role). This

method returns the total number of species, including the stoichiometric values, which was a problem for our new functions because we only wanted to know the number of substrates without the stiochiometric values. So we created a new method created in the file named above that is called getNumberOfSpecies() that overrides the mentioned problem.

```
CChemEqInterface::getNumberOfSpecies(CFunctionParameter::Role role)
```

At the time of creating the new functions based on the reaction, a boolean attribute was added to CFunction called *mAutogenerate* which will help us to identify at all times which are the original functions of COPASI and which are the dynamic functions. In the CFunction constructor, *mAutogenerate* is initialized to false by default. A setter and getter are defined for this boolean variable.

```
setAutogenerate(const TriLogic & autogenerate)
TriLogic & CFunction::isAutogenerate()
```

The new created functions are all dynamic. This means they all vary according to the substrates and modifiers of the selected reaction. For each of the three new functions, a static method has been created that receives by parameter the number of substrates and the number of modifiers of the reaction, which are obtained with the getNumberOfSpecies() described above. They generate various strings that will be merged to create the formula definition string that will be used in the returned CKinFunction. The construction of these strings is done through the use of *ostringstream* since it facilitate its construction. These added methods, create and set a new CKinfunction. This includes setting the string of the kinetic function, the attribute *mAutogenerate* and the reversibility that in the case of the dynamic functions is always false.

```
static CFunction * getPowerLaws(noSubstrates, noModifiers)
static CFunction * getSaturatingCooperative(noSubstrates,
noModifiers)
static CFunction * getSaturating(noSubstrates, noModifiers)
```

When a new dynamic function is created, it is stored in the global functions list. This means the generated functions may be reused in different reactions that are similar (reactions that have the same number of substrates and modifiers). To achieve this, we check the function names to know if the required function has been already created. The names of the functions are composed of the name of the function plus the number of substrates plus " s " plus the number of modifiers plus " m ".

```
static std::string getPowerLawsName(noSubstrates, noModifiers)
static std::string getSaturatingCooperativeName(noSubstrates,
noModifiers)
static std::string getSaturatingName(noSubstrates, noModifiers)
```

The last six methods are public, static and all are in the same file, called CAutogenerateFunctions. Since these methods are static, they can be called from all the points of the code without the need to create an instance of it, facilitating its use and leaving the code much cleaner. In this same file there is a get that returns a vector of strings with the names of the dynamic functions.

The CFunction returned by these static methods, are in fact of the type CKinFunction. This had to be done in order to be used in the user graphical interface application version.

```
CFunction * PowerLaws = new CKinFunction(Name)
```

In the image you can see that the Power Laws function has been created successfully for a substrate and zero modifiers. If function was created without the use of CKinFunction, in this tab that gives information about the function, we got a segment violation.

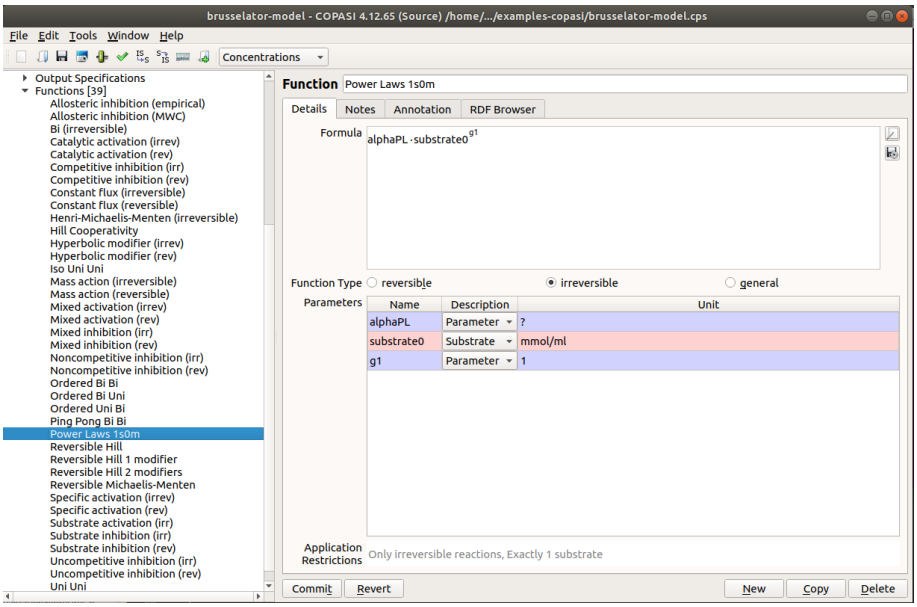

Fig. 2: Function description tab

The implementation of dynamic functions first involved finding where the rate law combobox was located in the code. Once the graphic file is found, we went to look for the C++ file which had the reference to the same combobox. Then we found that it was initialized with the result of the getListOfPossible-Functions() method of the CReactionInterface file. This method returns as a result a vector of the strings with the names of the suitable functions for that reaction. Once the function is selected another, setFunctionAndDoMapping() method is called from the same file. This method receives as a parameter a string corresponding to the name of the selected function in the combobox and performs a search on the general function vector, leaving an initialized CReac-tionInterface attribute with the pointer to that function to be able to apply the

function. The method that checks whether a function is suitable or not for a reaction is the isSuitable() and it receives as a parameter the number of substrates and products of the reaction and filters the functions as appropriate or not. In the case of dynamic functions, they are always suitable for all reactions. Therefore, we look at the value of the *mAutogenerate* attribute previously created, so that the dynamic functions are always in the combobox. At the time of filling the combobox for each of the dynamic functions it is checked if it has been previously created, so we will use that one instead of generate a new function. When a function is clicked, the method setFunctionAndDoMapping() calls the static methods of the CAutogenerateFunctions file and generate the corresponding new CFunction. In this way it will be sure that a dynamic function of each type will always appear. If it is created, it will be stored in the general function list.

As you can see in the code below, we check in the global function list vector for the names of the dynamic functions. We use a boolean variable to determine if the dynamic function generate entry will appear or not in the combobox. This entry is for generating a new function. If there is already a situable dynamic function, that one will appear. The shown code is for the Power Laws function but there is a similar one for all the other dynamic functions with the same function.

```
if(functionVector[i]->getObjectName().compare(
CAutogenerateFunctions::getPowerLawsName(mChemEqI.
getNumberOfSpecies(SUBSTRATE), mChemEqI.
getNumberOfSpecies(MODIFIER))) == 0){
  isaddPowerLaws = false;
}
```

You can see in the previous code that the static methods are used to generate the names of each function and also the method that returns a vector of strings with the names of the dynamic functions.

The following code shows how the setFunctionAndDoMapping() method has been implemented, specifically the Power Laws section. First we check if the string that arrives by parameter starts with “Power Laws”. If it starts with this string, we use the findLoadFunction() that allows us to get a CFunction by the means of its name. If the function is found, we use already stored function, so we reuse the function and we don’t generate a new function.

```
if (fn.compare("Power Laws") == 0) {

mpFunction = dynamic_cast<CFunction *>
           (CRootContainer::getFunctionList()->
           findLoadFunction(CAutogenerateFunctions::
           getPowerLawsName(mChemEqI.getNumberOfSpecies(SUBSTRATE),
           mChemEqI.getNumberOfSpecies(MODIFIER))));
if (mpFunction == NULL){
  CFunction * PowerLaws = CAutogenerateFunctions::
  getPowerLaws((mChemEqI.getNumberOfSpecies(SUBSTRATE)),
```

```
    (mChemEqI.getNumberOfSpecies(MODIFIER)));
    CRootContainer::getFunctionList()->add(PowerLaws,true);
    mpFunction = dynamic_cast<CFunction *> (PowerLaws);
    }
  }
```

The code of the Power Laws function is repeated for each of the dynamic functions named above.

Finally the option that has been implemented is that the functions that are autogenerated by our modification are set to be of the new Autogenerate type that we added. This is defined in the Evaluatetree file and by default all the static functions are predefined.

Algorithm 1 Power Laws auto generation workflow

1: User selects a reaction of the model
2: Combobox for function selection is generated:
3: $comboboxEntries \leftarrow functionList.getCompatible(selectedReaction).names()$
4: **if** Compatible with reaction Power Laws is not in comboboxentries **then**
5: $comboboxEntries \leftarrow$ "$PowerLawsautogenerateentry$"
6: **end if**
7: User selects a function in the combobox:
8: **if** User clicks the "Power Laws" autogenerate entry in the combobox **then**
9: Create a new Ckinfunction()
10: Generate the function string with substrates and modifiers based on the selected reaction
11: Add the new autogenerate function to the functionList and assign it to the selected reaction
12: Perform the function parameter mapping of the substrates and modifiers for the selected reaction
13: **end if**

The user clicks on the combobox it shows all the functions that are suitable for the selected reaction. If any of the dynamic functions exist, it is added to it and it works as a static function. Otherwise a string with the name of the dynamic function is added in the combobox in order to autogenerate that function. If the user clicks to autogenerate a function, a new CKinFinction is created by calling the static functions of the AutogenerateFunctions file which generates all the required attributes for it. The new function is added to the list of functions, so that it can be used for reactions with the same number of substrates and modifiers. Finally the mapping of all the substrates, modifiers, products and parameters of the reaction is done together with the new function. Each type of parameter have a different color according to COPASI rules. The mapping in the GUI is done by the means of introducing the parameter values, substrates, modifiers and products, opening a drop-down list with all species. By default the program has been made to take the same substrates and modifiers from the selected reaction.

4 Results

4.1 New Functionality

The final result has been to be able to use dynamic functions that previously in each reaction had to be manually inserted. You now have the option to use the functions Power Laws, Saturating Cooperative and Saturating directly without manual definition of these functions. Previously this was not possible.

In the following screenshot you can see how the COPASI is in its original version. The brusselator model has been selected and the Reaction 2 has been selected. There are only two suitable functions that are the Constant Flux and the Mass Action. Another detail that can be seen is the mapping of the function for this reaction. The Mass Action has 3 substrates: two of the species " X " and one of the " Y ". For the dynamic functions this mapping was not useful. A new mapping was created in which it only took the different species of substrates and modifiers of the reaction.

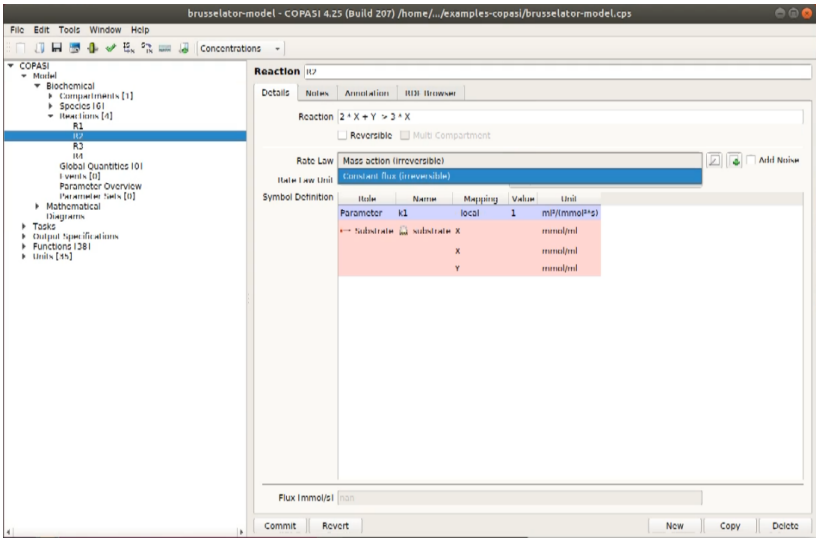

Fig. 3: Original COPASI version

The following capture corresponds to the modified COPASI version. In it you can see how the three dynamic functions are in the combobox ready to be used.

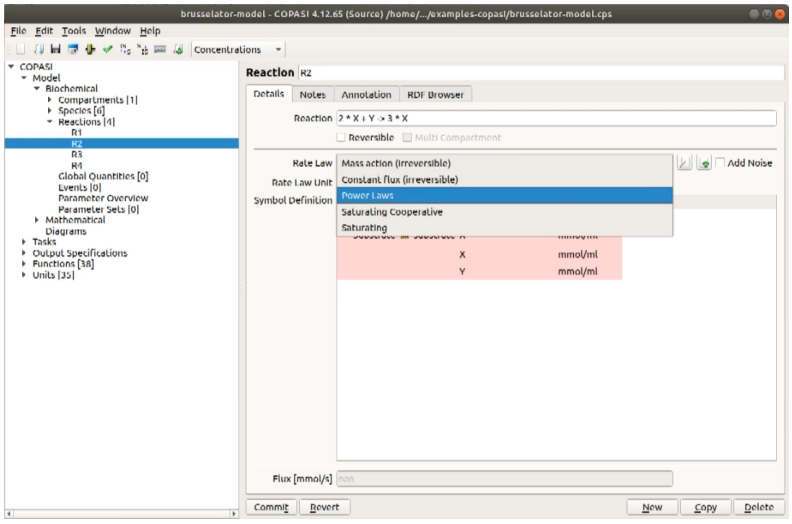

Fig. 4: Modified COPASI version

The mapping of a Saturating Cooperative with two substrates and three modifiers. With the new method that has been done is done correctly for both the substrates and the modifiers, setting by default the values of the selected reaction. If you want to change it is allowed to modify the substrates and modifiers to the loaded species in that model.

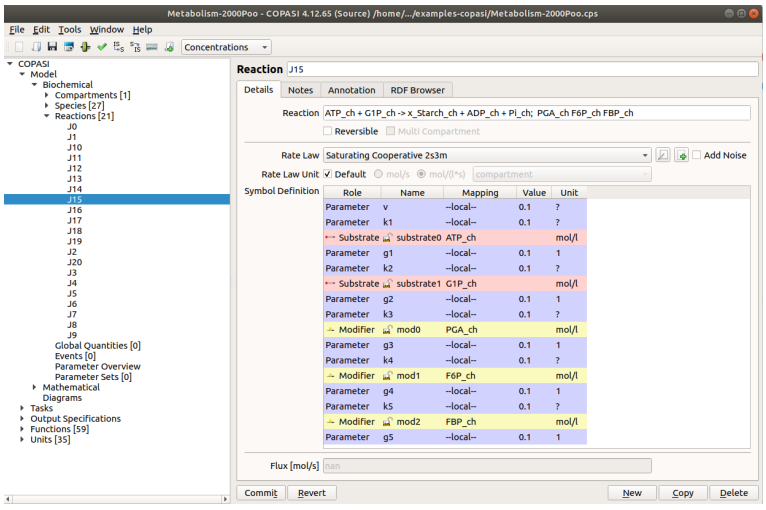

Fig. 5: Mapping of substrates and modifiers

As you can see in figure 4 that corresponds to the modified version at the time you just opened the program. You will see the 3 dynamic functions in the combobox without specifying the number of substrates and modifiers. Once you click on one of them you create the new function by assigning it to the reaction and the global function list.

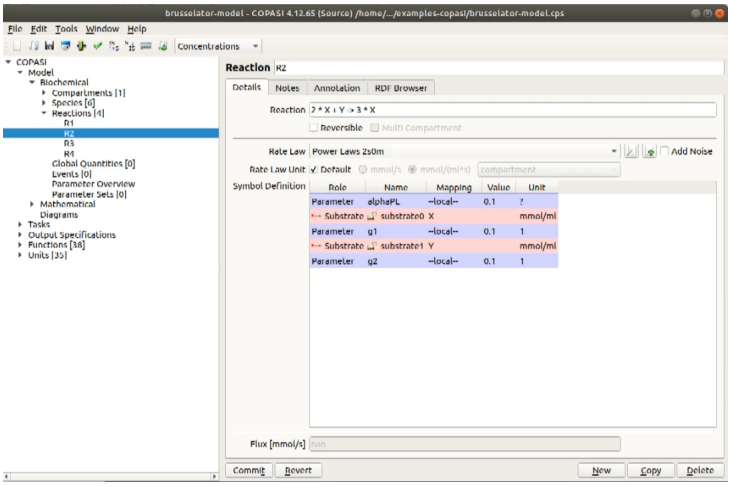

Fig. 6: Power Laws function selected

Once the new function has been added to the list of functions it can be reused if there is another reaction that has the same characteristics as the function of the list, thus no duplicates are created and there is always only one dynamic function of each type in the combobox.

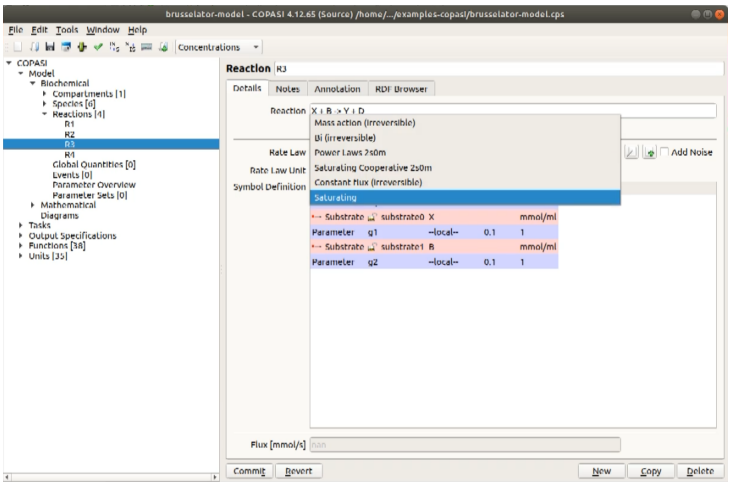

Fig. 7: Power Laws function reused

In the reaction that the reused function is applied, the parameters mapping is done again adjusting to the new reaction. If this process is carried out in two reactions that are in the same model, it does not cause any problem since it has been done taking this aspect into account at the reaction level.

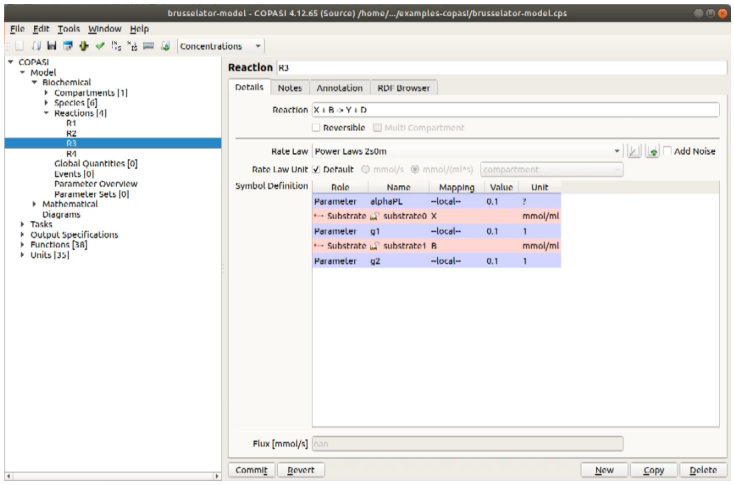

Fig. 8: Mapping of Power Laws in R3

4.2 Test case: Overall feedback in amino acid biosynthesis

To test the effectiveness of the new dynamic functions, we selected a model to be build from scratch in both original version of COPASI and the modified version. The selected model is the Overall feedback that can be found in the EasyModel web application. With this example we want to make a comparison and see if the implementation of dynamic functions in the COPASI is of use in a real case.

The first thing that has been done is to create the species that has the model. In each one of them we set its name and the initial concentration. In the first step, both the original and the modified version have to do the same steps, therefore the time has not varied between the 2 versions and it has been 92 seconds.

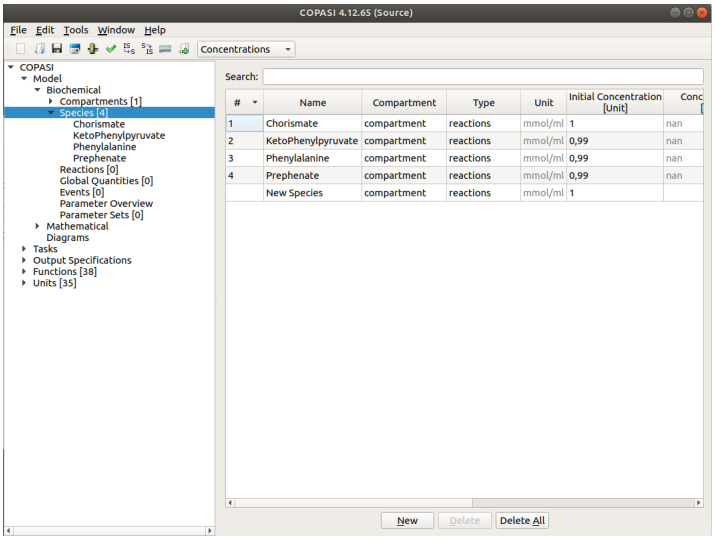

Fig. 9: Test: setting species

The second step is to create the functions that will be used in each of the reactions. For this model we must 2 version of the Power Laws: one with a substrate and zero modifiers and another with a substrate and a modifier. In the original version the two have been created by manually adding the formula and then manually doing the mapping of each parameter. In the modified version these steps do not have to be carried out, therefore the time inverted is 0 seconds while in the original version it is 225 seconds.

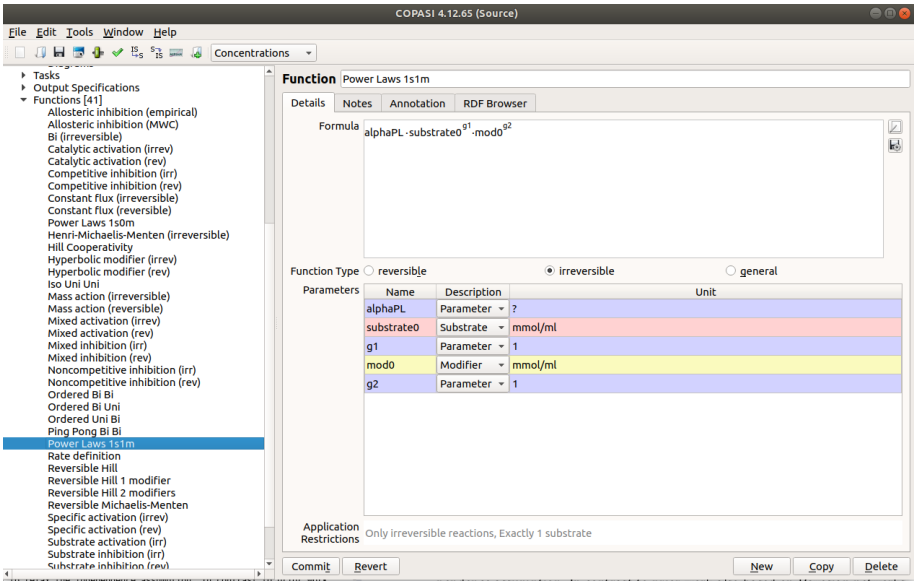

Fig. 10: Test: creating new functions

The third step is to create the model's reactions. In this case there are 4. In each reaction you have to add the chemical equation of the reaction, enter the name of the reaction and select the function for that reaction. In the original COPASI you have to look at the reaction and count the number of substrates and modifiers before selecting a function. In the modified version you just have to select Power Laws. In the original version it took 125 seconds to perform this step and in the modified 120 seconds.

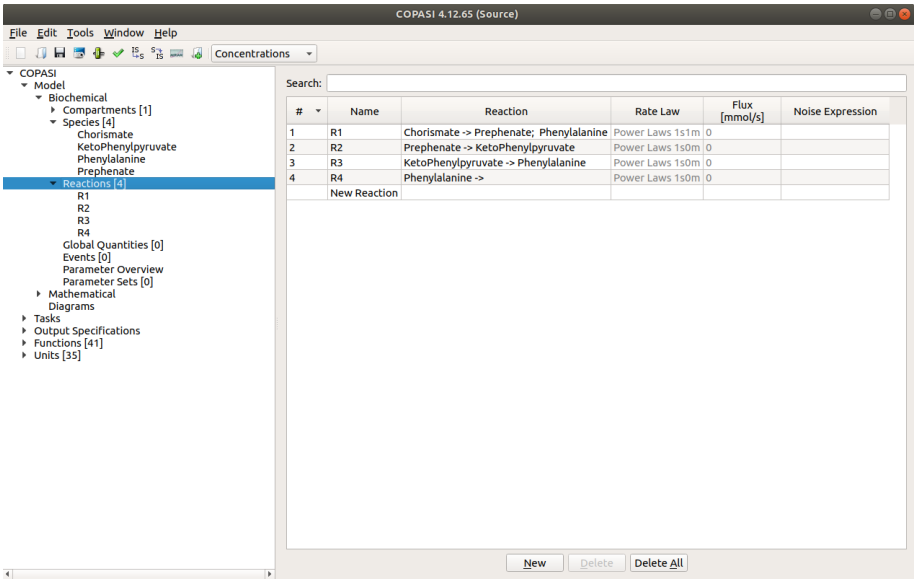

Fig. 11: Test: reactions with selected functions

In the following table a summary of the times has been made in each of the steps described above and the total time has been added to see more clearly the difference between the versions. Observing the result we can say that the modified version took approximately half of the time that the original version needed to create the Overall Feedback model from scratch.

	Species	**Functions**	**Reactions**	**Total**
Original	92	225	125	442
Modified	92	0	120	212

Table 2: Time comparison between the original and the modified version (in seconds)

5 Discussion, conclusions and future work

The work that has been done could be improved by using the Mass Action implementation style. This function has a producer that multiplies as many substrates as the reaction has. In order to use this kind of function, it would have helped to have a built-in numeric vector that stored a number for each of the substrates and modifiers.

In conclusion in this project have added three dynamic functions (Power Laws, Saturating Cooperative and Saturating) to COPASI. To achieve this, new methods have been created to help us implement dynamic functions in a code that was designed for static functions. The new functions are created and tailored for a specific number of substrates and modifiers. When they are created, they are added to the global list of functions, so it can be reused in another reaction. The way in which functions are created and reused makes the code efficient with respect to the original code and at the same time is easy to use since the

modified version maps each reaction in which a dynamic function is applied instantaneously, creating a feedback with the user and making the application usable.

Finally, in the future, dynamic functions using the implementation style of the Mass Action function could be implemented, and for this purpose, a vector of parameters should be implemented in the same way as that of the substrates and modifiers. Another improvement would be to add their Taylor calculation algorithms specially designed for the new dynamic functions.

References

1. Alves, R., Antunes, F., Salvador, A.: Tools for kinetic modeling of biochemical networks. Nature Biotechnology **24**(6), 667–672 (jun 2006). https://doi.org/10.1038/nbt0606-667, http://www.nature.com/articles/nbt0606-667
2. Alves, R., Vilaprinyo, E., Hernádez-Bermejo, B., Sorribas, A.: Mathematical formalisms based on approximated kinetic representations for modeling genetic and metabolic pathways. Biotechnology & genetic engineering reviews **25**, 1–40 (2008), http://www.ncbi.nlm.nih.gov/pubmed/21412348
3. Bartolome, J., Alves, R., Solsona, F., Teixido, I.: Easymodel web, https://easymodel.udl.cat
4. Ferreira, A.: Power law analysis and simulation (plas) (1996–2012), http://enzymology.fc.ul.pt/software/plas
5. Hoops, S., Gauges, R., Lee, C., Pahle, J., Simus, N., Singhal, M., Xu, L., Mendes, P., Kummer, U.: COPASI - A COmplex PAthway SImulator. Bioinformatics (2006). https://doi.org/10.1093/bioinformatics/btl485
6. Mendes, P., Hoops, S., Sahle, S., Gauges, R., Dada, J., Kummer, U.: Computational modeling of biochemical networks using COPASI. Methods in Molecular Biology (2009). https://doi.org/10.1007/978-1-59745-525-1$_2$
7. SBML: Software matrix web, http://sbml.org/SBML_Software_Guide/SBML_Software_Matrix
8. Sorribas, A., Hernández-Bermejo, B., Vilaprinyo, E., Alves, R.: Cooperativity and saturation in biochemical networks: A saturable formalism using Taylor series approximations. Biotechnology and Bioengineering (2007). https://doi.org/10.1002/bit.21316
9. Voit, E.O., D.I., Savageau, M.: The User's Guide to ESSYNS. Medical University of South Carolina Press (1989)